1995

Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++

Gerald Baumgartner

Vincent R. Russo

Report Number:
95-051

SIGNATURES: A LANGUAGE EXTENSION FOR IMPROVING TYPE ABSTRACTION AND SUBTYPE POLYMORPHISM IN C++

Gerald Baumgartner
Vincent F. Russo

Department of Computer Science
Purdue University
West Lafayette, IN 49707

CSD-TR-95-051
August 1995

Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++*

Technical Report CSD-TR-95-051

Gerald Baumgartner Vincent F. Russo
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

gb@cs.purdue.edu russo@cs.purdue.edu

August 11, 1995

Abstract

C++ uses inheritance as a substitute for subtype polymorphism. We give examples where this makes the type system too inflexible. We then describe a conservative language extension that allows a programmer to define an abstract type hierarchy independent of any implementation hierarchies, to retroactively abstract over an implementation, and to decouple subtyping from inheritance. This extension gives the user more of the flexibility of dynamic typing while retaining the efficiency and security of static typing. With *default implementations* and *views* flexible mechanisms are provided for implementing an abstract type by different concrete class types. We first show how the language extension can be implemented in a preprocessor to a C++ compiler, and then detail and analyze the efficiency of an implementation we directly incorporated in the GNU C++ compiler.

1 Introduction

The basic ideas of object-oriented design and programming are to group data structures, together with the functions operating on them, into new types, to encapsulate the data, and to keep interdependencies between types thus created as few as possible.

C++, as well as most other typed object-oriented languages, provides the *class* construct for defining and implementing types. In addition, *inheritance* is provided as a mechanism to reuse the code of existing classes and to define a subtype relationship between classes. The overloading of a single language construct, the class, for defining a type, for implementing a type, as the basis for code reuse, and as the basis for subtyping, not only limits the expressiveness of type abstraction and subtyping, but also the flexibility of inheritance for code reuse. To illustrate why this is the case, we first define some terminology and follow with examples of the limitations of classes.

1.1 Types

Semantically, a *type* can be viewed as a set of values [13]. In most traditional programming languages, such a set is characterized implicitly by providing an implementation for the type, e.g., using a primitive type the hardware provides or using a user-defined record type. However, there is another way to characterize a type: by specifying base elements, operations on the type, and axioms limiting the behavior of those operations.

*A slightly improved version of this paper appeared in *Software—Practice & Experience*, 25(8):863–889.

For example, a stack could be specified by `emptystack` as a base element, the operations `push`, `pop`, and `top` with the *function signatures*

```
stack emptystack ();
stack push       (T, stack);
stack pop        (stack);
T     top        (stack);
```

where `T` is the type of the stack elements, and the axioms

```
pop (emptystack ())       == ERROR
pop (push (elem, stack)) == stack
top (emptystack ())       == ERROR
top (push (elem, stack)) == elem
```

A type specified in this way is called an *abstract type*. In contrast, a type defined by an implementation is termed a *concrete type*. Since an abstract type lacks a representation, it must be implemented by a concrete type whose operations satisfy the function signatures and the axioms of the abstract type. For the purpose of this paper, we define *type abstraction* to be the process of abstracting over a preexisting or yet to be defined implementation, yielding an abstract type. Using abstract types allows a program to be written independent of the implementation of types. Implementations of abstract types, i.e., concrete types, can then be exchanged or added later without affecting the program. For example, a program written in terms of an abstract stack type can use any stack implementation, such as an implementation based on arrays or one based on linked lists.

Most object-oriented languages don't allow the definition of abstract types in this manner. Checking whether an implementation of an abstract type satisfies the axioms could not be done in a compiler since it is undecidable. In C++, the closest approximation is to define an *abstract class* with the implementations as subclasses. Since implementations are required to be subclasses of the abstract class, flexibility in structuring type hierarchies is lost. An example below demonstrates this point.

The C++ extension we present in this paper allows, instead, the definition of a *signature type*, an abstract type without axioms, *independent* of any class hierarchy. Having a language construct for defining a type separate from the construct used to implement a type allows greater flexibility in designing type hierarchies.

1.2 Subtype Polymorphism

Polymorphism is the potential for the same variable in a program to refer to values of different concrete types at different points in time. Cardelli and Wegner [13] distinguish four kinds of polymorphism: parametric polymorphism, subtype polymorphism, overloading, and coercion.

Some forms of *overloading* and *coercion* appear in nearly every programming language, with relatively strong support in C++. *Parametric polymorphism*, which allows functions to work over a potentially infinite range of argument types, is mostly found in functional languages, such as ML [27, 28] and HASKELL [17]. Generics in ADA [5] and templates in C++ [18, 33] provide a limited form of parametric polymorphism as well.

The form of polymorphism relevant to this paper is *subtype polymorphism*, or *subtyping* for short. It is found in some form in most strongly typed object-oriented languages. In subtype polymorphism, the different concrete types to which a variable can refer are limited to be *subtypes* (according to some subtype relationship between types) of the declared type of the variable.

Informally, a type t_1 is a subtype of a type t_2 if a value of type t_1 can be used wherever a value of type t_2 is expected. In C++, as in many other typed object-oriented languages, the subtype relationship is defined by the inheritance hierarchy, and 'subtype' is synonymous with 'subclass.' In object-oriented languages, the subtype relationship is seldom extended to builtin types and function types.

Having the subtype relationship based on inheritance can severely limit subtyping. To implement abstract classes, the implementation classes are typically made subclasses of the abstract classes. This requires

designing the inheritance hierarchy with implementation considerations in mind and, therefore, causes a loss of freedom in defining the subtype relationship. An example below demonstrates this point. It is possible to program around the problem by duplicating code or by writing forwarding classes, but both are cumbersome for the programmer.

In our language extension, we provide a mechanism to achieve subtype polymorphism based solely on the member function types (i.e., the signature, the structure, or the interface) of a class, independent of the class hierarchy. Subtyping defined this way is more general and, together with a separate language construct to declare a signature type, provides more freedom in defining type hierarchies than with classes alone. To make the language extension a conservative extension of C++, we use subtyping based on class interfaces only in connection with signatures. Although the extension is specific to C++, any statically typed object-oriented language can be extended in a similar way.

The next section further illustrates the problems that are caused by using inheritance for subtyping purposes. We follow with a section explaining the design of the language extension. Next, following some examples, we describe two implementations for translating signatures into C++ and assembly language, respectively, and analyze the run-time cost of the latter. All the type checking is done at compile time in both implementations. In the former implementation, the cost of calling signature member functions is two function calls, one of which is virtual, while in our GNU C++ implementation it is a few machine instructions in addition to the cost of only one virtual function call.

2 Limitations of Classes

In this section, we present some examples that demonstrate the limited use of classes for type abstraction and subtyping. The last example also shows that by overloading the use of inheritance, its possibilities for code reuse have to be restricted in order not to break subtyping.

2.1 X-Windows Object Manager

A practical example, illustrated in [21], shows that inheritance, as a substitute for subtype polymorphism, is not flexible enough in dealing with multiple implementations of an abstract type.

Imagine two libraries containing hierarchies of classes for X-Windows display objects. One hierarchy is rooted at `OpenLookObject` and the other at `MotifObject`. Further suppose all the classes in each hierarchy implement a `display()` and a `move()` member function. For the sake of argument, assume source code is not available for the two libraries, only header files and binaries are provided. The two implementations, therefore, cannot be modified to *retroactively* inherit from a common base class. Problems arise when when constructing a display list that can contain objects from *both* class libraries *simultaneously*.

One solution would be to build the list as a discriminated union (i.e., union plus tag field) of pointers or references to the base class of each hierarchy, to set the discriminant on each assignment and to select the proper reference each time an operation in invoked. The type tag is needed to determine whether to call `displayList[i].pMotifObject->display()` or `displayList[i].pOpenLookObject->display()`. The inelegance of this solution should be apparent.

In C++, as in other languages, multiple inheritance [32] can also be used to solve this problem. If an abstract class is constructed that defines the operations `display()` and `move()`, a set of new classes, each corresponding to an existing library class, can be constructed, such that each new class inherits both from the corresponding original library class and the abstract class. The implementation of the methods in the new classes can just forward the operation to the implementation in the original library class. In C++, the task of creating these classes can be simplified using templates [18, 33]. However, even with templates this option entails substantial software engineering costs. Building all the extra classes is tedious at best, and clutters the program name space with a superfluous set of new class names.

The difficulty in creating a display list that can contain both `MotifObjects` and `OpenLookObjects` is that C++ constrains the type of an object reference or pointer to a class, providing only one mechanism

(inheritance and virtual functions) to achieve the needed polymorphism. Consequently, even if every class in both libraries implements the `display()` and `move()` functions with the same interface, it cannot be expressed in C++ that this is the *only* requirement for addition to the display list and for manipulation of objects on the list.

2.2 Computer Algebra

An example from computer algebra similar to the one found in [8] demonstrates how a complex type hierarchy cannot be modeled and implemented by a class hierarchy. In computer algebra, the distinction between abstract types and concrete types arises naturally. Typical abstract types are `Group`, `Ring`, `Field`, or `EuclideanDomain`, while typical concrete types are `Integer`, `Fraction`, or `DistributedPolynomial`. To maintain these differences, abstract and concrete types should not be pressed into the same class hierarchy of an object-oriented language. Otherwise, code duplication is required.

Consider, for example, the abstract types `GeneralMatrix`, `NegativeDefiniteMatrix` and `OrthogonalMatrix`. Both negative definite matrices and orthogonal matrices are subtypes of general matrices since they have some functions, like `inverse()`, that are not available for general matrices. On the other hand, there might be several different implementations of matrices, such as two-dimensional arrays (`DenseMatrix`), lists of triples (`SparseMatrix`), or matrices in `BandMatrix` form. Assuming only single inheritance, pressing these implementations into the same hierarchy as the abstract types requires duplication of code to use the appropriate implementation for both negative definite matrices and orthogonal matrices. Two copies of the dense matrix implementation, `DenseNegativeDefiniteMatrix` and `DenseOrthogonalMatrix`, would be needed, as well as two copies each of the sparse matrix and band matrix implementations.

Code duplication could be avoided by using multiple inheritance. Each of the three implementation classes would then inherit from each of the three abstract classes. However, a problem arises when adding a subclass to one of the matrix implementations. Assume the class `PermutationMatrix` is to be added as a subclass of `SparseMatrix` in order to implement multiplication of permutation matrices more efficiently. Since a permutation matrix is positive definite, the new class can only be used as an implementation of `GeneralMatrix` and `OrthogonalMatrix`. No matter how the classes are structured, either code is duplicated, or `PermutationMatrix` is incorrectly allowed as an implementation of `NegativeDefiniteMatrix`.

Similar arguments have been presented in the literature to show that the `collection` class hierarchy of SMALLTALK-80 [20] is not appropriate as a basis for subtyping. While the problem does not arise directly in SMALLTALK-80, since it is dynamically typed, it becomes an issue when trying to make SMALLTALK-80 statically typed while retaining most of its flexibility. Proposed solutions include factoring out the implementation aspect of classes into prototypical objects [24] and factoring out the type aspect into interfaces [10, 14].

2.3 Doubly Ended Queues

Inheritance is not only an inadequate mechanism for achieving subtyping. If inheritance is used to define a subtype relationship, it can either make subtyping unsound or limit the flexibility of inheritance for code reuse.

An example similar to one in [30] illustrates this point. Consider two abstract types `Queue` and `DEQueue` (doubly ended queue). The abstract type `DEQueue` provides the same operations as `Queue` and in addition two operations for enqueuing at the head and for dequeuing from the tail of the queue. Therefore, `DEQueue` is a *subtype* of `Queue`.

For implementing `Queue` and `DEQueue`, the easiest way is to structure the inheritance hierarchy opposite to the type hierarchy. A natural choice for implementing a doubly ended queue is to use a doubly linked list. A queue implementation could then just inherit the doubly ended queue implementation and ignore, or hide, the additional operations.

For the type system to be sound it is not possible to define a subtype relationship based on inheritance and also allow to hide operations of a superclass [15]. Therefore, most object-oriented languages choose

instead to disallow inheriting only part of a superclass and to restrict the use of inheritance for code sharing to situations where there is also a subtype relationship.

3 Design of New Language Constructs

We turn now to describing our language extension and illustrating how it addresses the problems described above. The key language construct added to support type abstraction in C++ and to allow separating subtyping from inheritance is called a *signature*. Signatures, in this language extension, are related to types in RUSSELL [16], signatures in ML [25, 26], type classes in HASKELL [17], definition modules in MODULA-2 [37], interface modules in MODULA-3 [12], abstract types in EMERALD [9], type modules in TRELLIS/OWL [29], categories in AXIOM [22] and its predecessor SCRATCHPAD II [35, 36], and types in POOL-I [4].

In this section, we specify syntax and semantics of signatures, signature pointers, and signature references. We then show how signatures allow the introduction of subtype polymorphism into C++ independent of the inheritance mechanism. For the remainder of this section, we discuss aspects of the language extension that allow a smooth integration of signatures with the rest of C++.

3.1 Signatures

For simplicity, the syntax of signature declarations is nearly the same as that of class declarations. The differences are

- the keyword `signature` is used instead of `class` or `struct`,
- only type declarations, constant declarations, member function declarations, operator declarations, and conversion operator declarations are allowed, i.e., a signature cannot have constructors, destructors, friend declarations, or field declarations,
- the visibility specifiers `private`, `protected`, and `public` are not allowed, neither in the signature body nor in the base type list,
- signature base types have to be signatures themselves (similarly, a signature cannot be the base type of a class),
- the type specifiers `const` and `volatile` are not allowed for signature member functions, and
- storage class specifiers (`auto`, `register`, `static`, `extern`), the function specifiers `inline` and `virtual`, and the pure specifier `=0` are not allowed.

In other words, a signature contains only *public interfaces*.

The reason for not allowing the type specifiers `const` and `volatile` is that, semantically, they are storage location specifiers and are meaningless for member functions. The function specifiers `inline` and `virtual` and the pure specifier `=0` are only needed for classes where member function declarations serve to define both an interface and an implementation. For signature member function declarations they would be meaningless.

Like a class declaration, a signature declaration defines a new C++ type. For example, the signature declaration

```
signature S {
    typedef int t;
    t * f ();
    int g (t *);
    S & h (t *);
};
```

defines a signature type S with member functions f, g, and h.

Instead of using a signature, the type S above could have been defined just as well as an abstract class, i.e., a class containing pure virtual member function declarations [18]. The behavior of both implementations would be equivalent. However, if the type hierarchy becomes more complex, it can no longer be modelled precisely with a class hierarchy as shown in the earlier computer algebra example. Unlike when using abstract classes, a differently structured signature hierarchy, separate from the class hierarchy, can be constructed, enabling the creation of more complex type hierarchies and allowing the decoupling of subtyping and inheritance. Finally, while an abstract class cannot be retrofitted on top of an existing class hierarchy without recompiling all existing source files, signatures can easily be defined as type abstractions of existing classes. This both improves C++'s capabilities for reusing existing code and provides a better design tool for specifying abstract types independent of their implementation.

3.2 Signature Pointers and References

Since a signature type declaration does not provide enough information to create an implementation of that type, it is not possible to declare objects of a signature type. In order to associate a signature type with an implementation, a *signature pointer* or a *signature reference* is declared and assigned the address of some existing class object. Signature pointers and signature references can thus be seen as run time *interfaces* between abstract types (or signature types) and concrete types (or class types).

Consider the following declarations:

```
signature S { /* ... */ };
class C { /* ... */ };
C o;
S * p = &o;
```

To type check the initialization of the signature pointer p or an assignment to p, the compiler has to verify that the implementation C satisfies the interface S, or in other words, that the class type C *conforms* to the signature type S. The conformance check, which is defined below, also defines a *subtype* relationship. In this example, the signature of C is a subtype of S if and only if C conforms to S.

A signature pointer or reference can also be assigned to, or initialized from, another signature pointer or reference. In this case, the right hand side signature needs to conform to the left hand side signature, or, in other words, the right hand side signature has to be a subtype of the left hand side signature.

A signature pointer cannot be assigned to a class pointer without an explicit type cast since, in general, the class of the object pointed to by the signature pointer is unknown. What can be done is to implicitly convert a signature pointer to a pointer of type void*:

```
S *    p = new C;
void * q = p;
```

The result is a pointer to the class object, but without the type information. With an explicit type cast, a signature pointer can be assigned to a class pointer but, like casting down the class hierarchy, this is an unsafe operation. The same holds for signature references.

3.3 The Conformance Check

The *conformance check* is the type check performed for initializing or assigning to a signature pointer or a signature reference. There is no *run-time* penalty involved, the conformance check can be done at *compile* time.

To test whether a class C conforms to a signature S, it is necessary to compare the structures of C and S recursively. A class C is said to conform to a signature S if

- for every member function in S, there is a public member function in C with the same name and with conforming return and argument types,

- for every operator and every conversion operator declared in `S`, there is a corresponding public declaration in `C` with conforming return and argument types,
- for every type definition in `S`, there is a public type definition of the same name and conforming structure in `C`, and
- for every constant declaration in `S`, there is a constant declaration of the same name and of conforming type in `C`.

A class member function `C::f` conforms to a signature member function `S::f` if

- the type of every argument of `S::f` conforms to the type of the corresponding argument of `C::f` and
- the return type of `C::f` conforms to the return type of `S::f`.

As the base case of the recursive definition of conformance, every type conforms to itself. Operators and conversion operators are treated exactly like member functions, only the syntax for calling them is different. Field declarations as well as private or protected member functions and constructors in `C` are ignored. `C` can have more public member functions or types than specified in `S`.

For example, suppose the conformance of class `C` to signature `S` is being tested. Given signatures `T` and `U` and classes `D` and `E`, let `U` conform to `T`, let `E` be a subclass of `D`, and let class `D` conform to signature `T`. The signature member function

```
T * S::f (D *, E *);
```

can be matched with any of the following class member functions:

```
T * C::f (D *, E *);
T * C::f (D *, D *);
T * C::f (T *, E *);
T * C::f (T *, T *);
D * C::f (D *, E *);
E * C::f (D *, E *);
U * C::f (D *, E *);
T * C::f (D *, E *, int = 0);
```

Note that conformance of member functions is defined using contravariance [11] of the argument types and covariance of the result types. This makes the signature-based notion of subtyping more general than C++'s class-based subtyping.

If more than one member function of `C` conforms to a single member function of `S`, the one that conforms best is selected using a variant of C++'s algorithm for selecting the function declaration that best matches the call of an overloaded function [18].

If a single member function of `C` conforms to more than one member function of `S`, an error is reported. It is possible to relax this restriction by considering different matches of `C`'s member functions with `S`'s member functions and by selecting the best match according to some metric on signature types, but any such rule would be sufficiently complex to confuse users.

Every signature contains an implicit destructor declaration. This destructor is matched with the class's destructor if defined or with the default destructor otherwise.

For testing the conformance of one signature to another, the test is exactly the same as for testing the conformance of a class to a signature.

3.4 Signature Inheritance and Subtyping

Conformance between two signatures cannot, in general, be determined from the inheritance hierarchy of signatures. Using the conformance test for signatures, it is obvious that signatures can be in a subtype relationship without one inheriting from the other. The following shows that the converse can hold as well.

Signature inheritance is defined roughly as textual substitution. Declaring a signature `T` to inherit from signature `S` is equivalent to copying all declarations from `S` into `T`, while changing all occurrences of signature pointers and references of types `S*` and `S&` in `S`'s declarations into `T*` and `T&`, respectively. If `S` inherits from another signature `U`, the declarations of `U` are merged into `S` before the declarations of the resulting signature `S` are merged into `T`. This is the same definition as that of interface inheritance in [10]. For example, the declaration of `T` in

```
signature S {
    S * f (int, S &);
};

signature T : S {
    T & g (int, S *);
};
```

is equivalent to

```
signature T {
    T * f (int, T &);
    T & g (int, S *);
};
```

These semantics of signature inheritance are necessary to ensure that the signature conformance check does not depend on the hierarchy of signature declarations.

To motivate this definition of signature inheritance consider an example from computer algebra. Assume the two abstract types `AGroup`, of additive groups, and `Ring`, and an implementation of matrices, `DenseMatrix`. Signature `Ring` inherits from `AGroup`; the matrix class `DenseMatrix` conforms to both signature types.

```
signature AGroup {
    AGroup * add (AGroup *);
    // ...
};

signature Ring : AGroup {
    Ring * mul (Ring *);
    // ...
};

Ring * p = new DenseMatrix;
Ring * q = p->mul (p->add (p));      // p * (p + p)
```

If signature inheritance were defined like class inheritance, i.e., merely copying the declarations from the base signature, then the function call of `mul` above would not type check correctly. The result of the addition would have type `AGroup` instead of type `Ring`, which is expected by the multiplication member function.

Because of the chosen form of inheritance, however, the signature types `AGroup` and `Ring` are in no subtype relationship. Neither are `S` and `T` above. For type `T` to be a subtype of `S`, `T::f` would need to be a subtype of `S::f`. But, due to contravariance, that would require the second argument type of `S::f`, `S&`, to be a subtype of `T&`. Hence, `S` would need to be a subtype of `T`, which is impossible since `S` doesn't have a member function `g`.

If types related by inheritance are not recursive, i.e., if they don't contain references to themselves, the inheritance relation *is* identical to the subtype relation. Only recursive types require subtyping to be treated separately. Using the signature conformance check described earlier, arbitrary recursive types are disallowed

from being in a subtype relationship. It is possible, though, to extend subtyping to recursive types using an algorithm similar to the one in [3]. However, to guarantee that conformance remains sound, this would require an object-level encapsulation of conforming classes instead of the usual class-level encapsulation in C++. The algorithm to check whether an implementation adheres to object-level encapsulation is beyond the scope of this paper.

3.5 The Signature of a Class

As an alternative to the `signature` construct we also provide the `sigof` construct as found in [21]. Given a class `C`, the signature `sigof(C)` is constructed by duplicating all the public type definitions, constant definitions, and member function declarations from `C` into a new, anonymous signature. Field declarations and private and protected member functions are ignored. For example, the signature

```
signature S {
    typedef int t;
    t * f ();
    int g (t *);
    S & h (t *);
};
```

could alternatively be defined as the signature of a class with the appropriate public interface:

```
class C {
public:
    typedef int t;
    t x;
    t * f ();
    int g (t *);
    sigof (C) & h (t *);
private:
    t * foo (t *);
};

typedef sigof (C) S;
```

The result is equivalent to the previous declaration.

Since `t` is only a type abbreviation for `int`, the declaration of `C::t` could have been left out by replacing all occurrences of `t` in `C` by `int`. The resulting signature would still be the same. If `t` were defined by a local class, union, signature, or enumeration declaration, it couldn't be left out.

If class `C` is recursive, e.g., if an argument type or the return type of one of its member functions is of type `C*`, the resulting type in the signature will still refer to `C`. If class `C` refers to `sigof(C)`, the resulting signature will become recursive as in the example above.

Whether a type, constant, or member function is inherited or defined in class `C` directly is irrelevant for inclusion in `sigof(C)`. This ensures that the result of `sigof(C)` does not depend on the inheritance hierarchy of `C`.

The `sigof` construct can also be used to specify a base type in a `signature` declaration, to declare a signature pointer or reference, or in a cast expression:

```
signature T : sigof (C), U { /* ... */ };
sigof (C) * p = (sigof (C) *) new C;
```

Wherever a signature name can be used, a `sigof` expression can be used as well.

3.6 Default Implementations

So far only signatures containing member function *declarations* have been considered. However, we also allow a signature to contain member function *definitions* (i.e., declarations together with implementations), which are called *default implementations*. Consider, for example, the signature

```
signature S {
    int f  (int);
    int f0 () { return f (0); };
};
```

Now a class `C` does not need a member function '`int f0 ()`' in order to conform to to `S`. If there is a member function `C::f0` of the right type, it will be used. If `C::f0` is not defined or is of the wrong type, the default implementation `S::f0` is used instead.

When specifying an abstract type, signature default implementations allow a prototype implementation for some member function to be written. Such a prototype both serves to specify the behavior and allows testing to start before a corresponding (and likely more efficient) class member function is written, i.e., it serves a similar purpose as a member function implemented in an abstract class.

Since it must be known at compile time whether a signature member function has a default implementation, all default implementations need to be defined inside the signature declaration. If a definition outside the signature such as

```
int S::f0 () { return f (0); };        // illegal
```

were allowed, it wouldn't be know until link time which signature member functions have default implementations and which ones do not. This would prevent performing the conformance check at compile time.

Another consequence of allowing default implementations is that they introduce one case in which the program cannot be type-checked fully at compile time. The problem arises when assigning a signature pointer of signature type `T` to a signature pointer of signature type `S`, where `T` contains a default implementation for member function `f` but `S` only contains a declaration of `f` without default implementation. Since it is not know at compile time whether the default implementation of `T::f` is actually used, a run-time test for it must be generated. Consider

```
signature S {
    int f ();
};

signature T {
    int f () { return 0; };
};

int foo (T * p)
{
    S * q = p;

    /* ... */
}
```

In the function `foo` above, it cannot be known whether `p` will use `T`'s default implementation or not. If the default implementation is used, a run-time type error in the assignment to `q` will occur. Using `T`'s default implementation when calling `q->f()` is not an option, since it would violate the static scoping rules of the language.

Note that this is the only case where a run-time type check is necessary, in all other cases conformance can be fully checked at compile time. To warn of the possibility of a run-time type error, the compiler should print a warning message when generating the run-time test.

An alternative would be to make the conformance test stricter so that two signature member functions only conform if neither one has a default implementation or if both inherited their default implementation from the same supertype. We feel that for most practical purposes, this would unnecessarily restrict the subtype relationship, but it might be useful to have a compiler *option* that makes the compiler print an error message instead of generating a run-time type check.

3.7 Constants

In the definition of the conformance check, it was mentioned that a signature can contain constant declarations. Conceptually, constants in signatures are treated similarly as nullary signature member functions. In particular, unlike constant declarations elsewhere, they do not need to be initialized. For example, for a class `C` to conform to the signature

```
signature S {
    const int n;
};
```

it has to have a public declaration of constant `n`. The value of `C::n` can then be accessed through a signature pointer as in the following example.

```
class C {
public:
    const int n = 17;
};

S * p = new C;
int i = p->n;
```

The variable `i` above gets set to the value `17`. If the constant `n` had been replaced by a nullary member function in both the signature and the class, with the class's implementation returning the constant value `17`, the behavior would be the same. The implementation of constants, however, is more efficient.

Similarly, initialized signature constants could be viewed as analogous to constant nullary functions with a default implementation. The value of the class's constant would then override the value of the signature's constant. Such an implementation would indeed be possible, but it wouldn't allow code such as

```
signature S {
    const int n = 17;
    typedef int[n] array;
    int f (array);
};
```

since the value of `n` would not be known at compile time. To make the value of *initialized* signature constants available at compile time, it is required that the value of the constant is the same in both the class and the signature. This restriction allows initialized signature constants to be used for defining data structures.

3.8 Opaque Types

Signature member functions typically don't have a default implementation and signature constants don't need to be initialized. By analogy, consider what happens if types in a signature are only declared and are defined in conforming classes. Such types, called *opaque* types, would allow programming is a style similar as in MODULA-2 or in the stack example given in the introduction. Opaque types can be used to hide implementation aspects from users of a signature in a similar way as the `private` keyword hides part of a class implementation. Experience will tell whether opaque types prove to be useful in practice.

To declare an opaque type, the typename is declared using the `typedef` construct without specifying a type expression. For example, the type `S::t` in

```
signature S {
    typedef t;
    t mk_t ();
    t foo  (t, t);
};
```

is an opaque type.

To call the member functions of the signature above, the compiler must be able to calculate the amount of memory to allocate for an object of type `S::t`. Since the definition of the opaque type is not available when compiling `S`, all uses of opaque types must be made to take the same amount of memory by requiring them to be implemented as pointer types. For example, the following class conforms to signature `S` above:

```
class C {
public:
    typedef int * t;
    t mk_t ();
    t foo  (t, t);
};
```

If the type `C::t` were not a pointer type, the class would not conform to the signature.

As mentioned earlier, since a normal `typedef` declaration in a signature is only a type abbreviation, it is not necessary for a conforming class to use the same `typedef` statement. An opaque type declaration, by contrast, is not a type abbreviation but defines a type. A conforming class, therefore, needs to contain a `typedef` specifying the implementation of the opaque type.

A signature that contains opaque type declarations can only be implemented by one class at a time. Otherwise it would introduce a loophole in the type system. Suppose class `D` conforms to `S` but implements `D::t` differently. If both `C` and `D` were allowed to coexist as implementations of the abstract type `S`, as in

```
S *  p = new C;
S *  q = new D;          // illegal
S::t x = p->foo (p->mk_t (), q->mk_t ());
```

it would be impossible to catch the type error in the second argument of `foo`. To test that globally only one class was used to implement `S`, linker support is needed to test that at most one signature table was generated for signature `S`. On a per file basis, it can be done in the compiler.

3.9 Views

For the conformance check it was required that signature member functions have to be matched by class member functions of the same name and the same type. Often it is desirable to allow a class member function to have a different name. The same holds for constants and types.

For example, suppose that in the X-Windows object manager example the member function to display a window on the screen is called `display()` in `OpenLookObject` but `show()` in `MotifObject`. To build a display list of objects from both hierarchies, it is necessary to rename the member function of one of these hierarchies.

To rename class member functions, or to *view* a class to be an implementation of a signature type in different ways, we provide the following syntax:

```
S * p = (S *; foo = bar) new C;
```

In this cast expression, the signature member function `foo` is associated with the class member function `bar`. The same effect can be achieved using the following variant of the `sigof` construct:

```
typedef sigof (C; foo = bar) T;
T * p = (sigof (C) *; foo = bar) new C;
```

The same syntax can be used for renaming constants and types.

For renaming multiple member functions, renaming pairs are separated by commas. To allow swapping of member function names, conceptually, the renaming operations are performed in parallel. This allows, for example, to view a rational number class as an implementation of the signature type `Group` in two different ways, as a multiplicative group and as an additive group, without having to write additional interface code.

For overloaded member functions, no special syntax is provided to selectively rename member functions depending on their return and argument types. While this wouldn't be difficult to implement, the syntax of views would become excessively complicated. Instead, all member functions with the same name are renamed the same.

A similar renaming mechanism can be found in VIEWS [1, 2], an experimental computer algebra system written in SMALLTALK, or in the algebraic specification language OBJ3 [19].

4 Examples

In this section, we present three examples that show a typical use of a signature with multiple implementations, how constants can be used to encode semantic information in a signature, and how signatures integrate with templates. In addition, we contrast the use of opaque types with the traditional C++ programming style.

4.1 Heterogeneous Collections of Objects

This example demonstrates how objects from two unrelated classes can be combined in a heterogeneous data structure. Assume that all that is required for an object to be included in a data structure is that it's class contains a nullary public member function `text` that returns a string. This requirement can be specified with the signature declaration

```
signature S {
    char * text ();
};
```

Since the two implementations of the signature type `S` don't need to be related by inheritance, they can easily be developed separately. They only need to *conform* to the signature, i.e., the signatures of the classes are structural subtypes of `S`.

```
class C {
    char * str;
public:
    C ()           { str = "Hello "; };
    char * text () { return str; };
};

class D {
public:
    char * text ()      { return "World."; };
    char * text (int i) { return "@#$%*!"; };
};
```

The fact that the member function `D::text` is overloaded, doesn't matter. As long as class `D` contains a member function '`char * text ()`,' it conforms to signature `S`.

Now a heterogeneous collection can be constructed by defining a data structure containing signature pointers of type `S*`.

```
#include <iostream.h>

int main ()
{
    S * p[2] = { new C, new D };

    cout << p[0]->text () << p[1]->text () << "\n";

    return 0;
}
```

When run, the above program will print the string "`Hello World.\n`".

4.2 Properties

Other languages, such as POOL-I [4] or AXIOM [22], offer language constructs for defining *properties* of signature and implementation types. These properties are used as a shorthand for axioms written in first-order logic or equational logic to encode semantic information as part of the specification of an abstract type. Constants provide a simple and efficient mechanism to implement properties without requiring additional syntax.

For example, an integer stack signature with the property `LIFO` could be defined as follows:

```
#define   LIFO       const int lifo_p = 0;

signature Int_Stack {
    typedef stack;

    stack emptystack ();
    stack push       (int, stack);
    stack pop        (stack);
    int   top        (stack);

    LIFO;
};
```

For comparison with the example in the introduction of this paper, this signature is written using an opaque type to represent the implementation type, i.e., in the style of an abstract data type definition. The property `LIFO` is intended to be an abbreviation for the four axioms given in the introduction. Note, however, that the use of properties does not depend on using this programming style.

In order to conform to the above signature, a class needs to contain the same property declaration in addition to member functions specified in the signature. The opaque type needs to be implemented as a pointer type, in this example as a pointer to a list element.

```
#define properties public

class Int_Stack_as_List {
private:
    struct  elem;

public:
    typedef elem *   stack;

    stack emptystack ();
```

```
    stack push      (int, stack);
    stack pop       (stack);
    int   top       (stack);

properties:
    LIFO;

private:
    struct elem {
        int   value;
        stack next;
    };
};
```

The intended meaning of a certain property defined in a class is that the operations of the class have been verified to satisfy the axioms represented by the property. That is, the operations have been certified to have the expected behavior.

4.3 Signature Templates

Like classes, signatures can be parameterized by writing signature templates, which allow the definition of generic abstract types. For example, a stack type parameterized by the type of its elements could be defined using the following signature template:

```
template <class Elem> signature Stack {
    void push (Elem);
    void pop  ();
    Elem top  ();

    LIFO;
};
```

This signature is written without using an opaque type, i.e., in the traditional C++ style. The private fields of an implementation class play now the role of the opaque type. As in the previous example, the property `LIFO` is used to specify the semantics of a stack. A possible implementation could be

```
template <class Elem, int N> class Stack_as_Array {
private:
    Elem contents[N];
    int  sp;

public:
    Stack_as_Array () { sp = 0; };
    void push (Elem);
    void pop  ();
    Elem top  ();

properties:
    LIFO;
};
```

To build a stack object, the templates must be instantiated. For example, let `Symtab` be an implementation of symbol tables. With the signature pointer declaration

```
Stack<Symtab> * stk = new Stack_as_Array<Symtab, 100>;
```

a stack of at most 100 symbol tables would be created.

It is important to note that templates don't introduce any additional complexity in the conformance check. In this example, both the signature template and the class template are instantiated first. The conformance check is then performed on the instantiated signature and class types.

5 Possible Implementations

In this section, we first outline a scheme for translating signature constructs directly into C++ classes. Using this method it is possible to write a compiler preprocessor, say `cfrontfront`, that translates from C++ with signatures into C++ without signatures. We then present our GCC [31] implementation, which is designed to minimize the run-time overhead. The implementation only modifies GCC's C++ front end, `cc1plus`; the technique is independent of the compiler and, e.g., could be used to implement signatures in the AT&T `cfront` compiler as well. Finally, we discuss some of the design alternatives and possibilities for optimization.

5.1 Translation into C++

The main idea of implementing signature pointers is to treat them as run time interfaces to class objects. In this implementation, signature pointers themselves are instances of some interface class and require twice as much memory as a regular C++ pointer.

Consider the declarations

```
signature S {
    int f ();
    int g (int, int);
};

S * p = new C;
```

and assume `C` conforms to `S`. For the signature declaration itself no code is generated, it is considered a type declaration only.

To make `p` an interface from abstract type `S` to concrete type `C`, an interface class `S_C_Interface` could be generated with `p` as an instance as follows:

```
class S_C_Interface {
    C * optr;
public:
    S_C_Interface (C * x) { optr = x; };
    int f ()              { return optr->f (); };
    int g (int x, int y)  { return optr->g (x, y); };
};

S_C_Interface p = new C;
```

When `p` is initialized, the instance variable `optr` is initialized to point to the object assigned to `p`. The member functions of the interface class only redirect to the appropriate member functions of class `C`.

However, the signature pointer `p` can now only be used to point to objects of class `C`. To make signature pointers point to arbitrary objects, an additional indirection is needed. This can be achieved by compiling the declaration of the signature `S` into an abstract class `S_Pointer`.

```
class S_Pointer {
public:
```

```
    virtual ~S_Pointer ()     = 0;
    virtual int f ()          = 0;
    virtual int g (int, int) = 0;
};
```

For building the interfaces between the signature and conforming classes a template class `S_Interface` is generated as public subclass of `S_Pointer`.

```
template <class T> class S_Interface : public S_Pointer {
    T * optr;
public:
    S_Interface (T * x)     { optr = x; };
    ~S_Interface ()         { delete optr; };
    int f ()                { return optr->f (); };
    int g (int x, int y)    { return optr->g (x, y); };
};
```

This template class is then instantiated with some class `C` to build the object interfacing `S` and `C`.

Signature pointers can now be implemented as pointers to instances of `S_Interface<C>` for some class `C`. That is, the declaration

```
S * p = new C;
```

would be translated to

```
S_Pointer * p = new S_Interface<C> (new C);
```

This has the advantage that it is very easy to implement in a preprocessor for a C++ compiler, but it requires interface objects to be allocated on the heap.

A more efficient solution is to use the interface object directly as signature pointer, i.e., to generate the code

```
S_Interface<C> p = new C;
```

With this implementation no heap allocation of interface objects is needed, but instead the preprocessor must be modified to make p behave as if it were a pointer to an object of type `S_Pointer` rather than a structure of type `S_Interface<C>`.

A signature pointer implemented thus requires the storage of two normal C++ pointers, namely the pointer to the class object, `optr`, and the pointer to `S_Interface`'s virtual function table. Signature references are implemented exactly the same way.

For implementing default implementations, a flag in the interface class `S_Interface` is needed that tells us whether the default implementation is used or whether a member function was provided by the class. The corresponding member function of the interface class then, depending on the value of the flag, either calls the class member function or executes the code of the default implementation.

Constants without initialization can be implemented by generating a public variable declaration in the interface class and initializing it with the object's constant value in the constructor.

5.2 Implementation in GCC

The main disadvantage of the above implementation is the overhead of an additional function call when calling one of `C`'s member functions. To make signature member function calls more efficient, we instead make the interface object a table of addresses of `C`'s member functions together with some flags.

Unfortunately, with this implementation it is only possible to achieve *strict conformance* between a signature and a class. For a class member function to conform to a signature member function, it needs to have the same number of arguments, the same return type instead of a conforming return type, and the same

argument types instead of argument types the argument types of the signature member function conform to. The problem is that in the more general case, the arguments and/or the return value may need type conversion when calling a signature member function. Since in the implementation described here there is no place to keep the conversion code, only strict conformance can be achieved. In the next section, we explain how this restriction can be removed.

Simplified Version

Ignoring default implementations, classes with virtual member functions, and multiple inheritance of classes for now, we internally translate the declaration of signature `S` above into the equivalent of

```
struct S {
    const void * _S_destr;
    const int    (S::*_f) ();
    const int    (S::*_g) (int, int);
};
```

The field `_S_destr` represents the implicitly declared destructor of signature `S`.

As in the previous solution, we make a signature pointer an object containing a pointer to the class object and a pointer to an interface object, the *signature table*.

```
struct S_Pointer {
    void *    optr;
    const S * sptr;
};
```

The structure `S` above is used as the type of signature tables for signature `S`.

For initializing a signature pointer, such as

```
S * p = new C;
```

a signature table must be generated and initialized if it doesn't exist already. The signature pointer structure is initialized with a default constructor call as follows:

```
static const S S_C_table = { &C::~C, &C::f, &C::g };
S_Pointer p = { (void *)new C, &S_C_table };
```

For an assignment, such as

```
p = new C;
```

or for passing an object to a signature pointer parameter, a compound expression must be generated. In the case of the assignment, the code generated is

```
static const S S_C_table = { &C::~C, &C::f, &C::g };
( p.optr = (void *)new C,
  p.sptr = &S_C_table,
  p
);
```

The signature table `S_C_table` takes on a role similar to that of the virtual function table in C++. The instance variable `sptr` of a signature pointer corresponds to the virtual function table pointer `vptr` in an instance of a class with virtual functions.

Initializing the signature table `S_C_table` requires casting `C::~C`, `C::f`, and `C::g` to member functions of `S`. This has to be done in the compiler front end, since C++ doesn't allow casting to a member function type.

Since signature tables are static and constant, only one signature table per signature-class pair in each file needs to be constructed, instead of one signature table per signature pointer assignment.

To translate a function call such as

```
int i = p->g (7, 11);
```

we dereference p's sptr and call the function whose address is stored in the field _g, i.e., C::g. The optr has to be passed as the first argument, so that C::g gets a pointer to the right object passed for its implicit first parameter called this.

```
int i = p.sptr->_g (p.optr, 7, 11);
```

In cases where the value of p->sptr is known at compile time, this can be optimized to call C::g directly.

Similar as in the translation to C++, uninitialized constants are implemented by generating a field declaration in the signature table type (i.e., in struct S) and initializing it with the class's constant value in the default constructor for the signature table.

Implementation Details

Above only interfaces to classes with non-virtual member functions were considered. The address of a virtual member function cannot be stored in the signature table since the address is not known until run time. Similarly, it is unknown at compile time whether to use an object's member function or a default implementation. To allow for all possibilities it is necessary to add two flags to each member function pointer in the signature table. One of the flags indicates whether the address of a virtual member function needs to be looked up at run time; in this case, instead of a pointer to the function, the offset into the virtual function table is stored together with the offset into the object at which to find the pointer to the virtual function table. The second flag indicates whether the function pointer points to a default implementation instead of a class member function; in this case, the signature pointer itself, instead of the optr, has to be passed as the implicit first argument when calling the function.

To deal with classes that are defined using multiple inheritance it is also necessary to store the offset that needs to be added to this for inherited member functions. Instead of being just a function pointer, a signature table entry now looks as follows:

```
struct sigtable_entry_type {
    short  code;
    short  offset;
    union {
        void * pfn;
        struct {
            short vptroff;
            short vtbloff;
        };
    };
};
```

The code field contains the flags explained above, offset contains the value that needs to be added to this. The field pfn contains a pointer to the code for a non-virtual member function or a default implementation. For a virtual member function the field vptroff contains the offset into the object to find the virtual function table and vtbloff contains the offset into the virtual function table. These two offsets occupy the same storage space as pfn.

For calling a signature member function, the compiler needs to generate a conditional expression that tests the appropriate flags before the call of the class's member function.

For assigning a signature pointer to another signature pointer, all that is required is to simply copy the optr and sptr fields if the signature pointers are of the same type and the LHS signature pointer is in local

scope. If the RHS signature is a subsignature of the LHS signature, it is possible to let the LHS `sptr` point into the RHS signature table at an offset determined by the signature hierarchy. If they are not related by signature inheritance or when assigning to a non-local signature pointer, the corresponding fields of the signature table have to be copied and the LHS signature table needs to be allocated in the same scope as the LHS signature pointer instead of in static memory.[1] Pointers to default implementations, in general, cannot be copied but might cause a run-time type error to be reported.

5.3 Design Alternatives

Thunks

As an alternative to using member function pointers in the signature table and to test flags to determine how to call a member function, the signature table could contain (pointers to) pieces of code, or *thunks*, that set the argument for the `this` parameter correctly and then branch to the class member function. Such an implementation is described in [7].

Advantages of using thunks would be that there would be no overhead of testing the flags in a signature member function call, and that no zero offset would need to be added to the `this` pointer in case of single inheritance. Instead, thunks would simply contain the appropriate code. Similarly, thunks are used in some implementations of virtual function tables to reduce the overhead in the single inheritance case.

A big advantage, though, would be that code for converting argument types could be included in a thunk. For converting the return type either a second thunk that does nothing else could be used, or a thunk could call, instead of branch to, the class member function using a light-weight function call sequence.

By making signature tables contain thunks the conformance check could, therefore, be implemented correctly, i.e., we would not be limited to strict conformance. There would be no run-time penalties compared to our implementation in GCC if a signature member function doesn't require conversions. On the contrary, by not having to test the `code` field, a few instructions would be saved. In the common case of calling a non-virtual member function with a zero offset for the `this` pointer, the thunk could be eliminated altogether by branching to the member function directly. The only disadvantage of using thunks is that it requires generation of low-level code, which complicates or even prohibits its use in a compiler that generates C code, such as AT&T's `cfront` compiler.

Signature Table Management

As with virtual function tables, in a system consisting of more than one object file the executable can contain several copies of a signature table since they are statically allocated in each compilation unit. Additional duplication of signature tables can be caused by assigning one signature pointer to another as discussed earlier.

There are two possibilities to decrease the number of tables. One is to provide a signature table manager in the run-time system that allocates memory for the signature tables when starting the program. The other possibility is to make signature tables external and to have linker support to eliminate identical tables. Both the signature table manager and the link-time table comparison could be implemented efficiently using the type matching scheme described in [23].

If a signature table manager is used and type information of objects is available at run time (e.g., using `typeid` [34]) the signature tables for virtual subclasses can be generated at run-time, eliminating the need for double indirection when looking up the address of a virtual member function and eliminating copying of signature tables when assigning one signature pointer to another.

[1] This complexity is necessary to correctly handle assignments to signature pointers in recursive functions. In the absence of recursion or when the signature hierarchy is designed such that the RHS signature pointer is of a subsignature type, all signature tables can be statically allocated. For more efficient implementations see [7].

6 Cost Analysis

For the purpose of the following discussion assume that a pointer can be stored in a machine word. The memory required for storing a signature table is then the memory needed for constants declared in the signature plus two words per table entry, i.e., two words for each member function, member operator, or conversion operator declared in the signature plus an additional two words for the destructor. In the absence of the pathological case discussed earlier, signature tables can be stored in statically allocated memory.

The space needed for a signature pointer or signature reference is also two words, one word for each of the `optr` and `sptr` fields. This makes a signature pointer or reference require twice the amount of memory needed for a normal C++ pointer.

The time needed for assigning to a signature pointer or reference is twice the time needed for assigning to a normal C++ pointer. The only exception is when one signature pointer or reference is assigned to another one, and the LHS signature is neither equal to the RHS signature nor an ancestor in its inheritance hierarchy. In this case, it is necessary to initialize the LHS signature table at run time, which requires copying two words for each table entry. Furthermore, if the RHS signature has a default implementation that is not also a default implementation of the LHS signature, is it necessary to test whether this default implementation is used and report a run-time type error if it is.

When calling a class member function through a signature pointer or reference, the `sptr` field needs to be dereferenced first to get the corresponding signature table entry. Then a test, whether a virtual function is being used, must be performed and, depending on the result, either a call to the class member function is made directly, or a virtual function lookup and call is performed. In either case, the `optr` value is passed as the argument for `this`. The overhead for calling a class member function through a signature pointer is roughly the same as the overhead for a virtual function call plus the cost of one test and dereferencing the `optr`. If a default implementation might be called an additional test instruction is needed.

7 Conclusion

In this paper, we discussed the limitations of class inheritance for constructing complex type hierarchies and argued that different mechanisms should be used for implementing subtype polymorphism and code reuse. We proposed language constructs for specifying and working with signature types. These constructs allow us to decouple subtyping from inheritance.

The result, C++ with signatures, has a type system related to those of several other modern programming languages. Similarly as in ML [25, 26], MODULA-2 [37], and MODULA-3 [12], signatures in C++ allow a clean separation of specification and implementation. However, ML and MODULA-2 only have modules and no classes, while MODULA-3 has both classes and modules but provides interfaces for modules only and not for classes. RUSSELL [16] and HASKELL [17] have notions related to signatures, but both lack classes. EMERALD [9] has first-class types instead of classes, and TRELLIS/OWL [29] has a class hierarchy in which only type information but no implementation is inherited. Signatures in C++ come closest to categories in AXIOM [22] and types in POOL-I [4]. But AXIOM is an abstract data type language, and POOL-I lacks overloading and private and protected member functions and fields. Also, while categories and domains in AXIOM as well as types in POOL-I are first class, signatures and classes in our C++ extension are not. This makes the type system slightly less expressive but it allows a more efficient implementation and permits complete type checking at compile time. To make up for the loss in expressiveness, like MODULA-3 we provide structural subtyping.

After discussing a straightforward translation of the new language constructs to somewhat inefficient C++ code, we described how to directly translate them into efficient assembly language and analyzed the performance of the latter. We also presented some promising possibilities for optimizing signature member function calls, which need further investigation.

Signature types are presented as a conservative extension of C++, but the concept would apply equally well to any statically typed object-oriented programming language. If a language with signatures and classes were designed from the ground up, there wouldn't be any need for a subtype relationship defined

by inheritance in addition to the subtype relationship defined by the conformance check. Similarly, virtual member functions would not be needed, signature member functions provide the desired polymorphism instead. With subtyping thus decoupled from inheritance, it would be possible to change the semantics of inheritance and make it more versatile for code reuse, e.g., by allowing to inherit only parts of superclasses. For pragmatic reasons, however, such drastic changes are undesirable for an extension of C++, as they would affect the behavior of existing programs.

Availability

Parts of the language extension have been implemented in the GNU project C++ compiler [6] as a compiler extension. The implementation is included in versions GCC-2.6.0 and higher.

Acknowledgements

We would like to thank Konstantin Läufer and Michal Young for reading parts of the paper and providing many valuable comments, and Andy Muckelbauer for numerous discussions about the implementation. William Cook's comments resulted in a greatly improved presentation.

References

[1] S. Kamal Abdali, Guy W. Cherry, and Neil Soiffer. An object-oriented approach to algebra system design. In Bruce W. Char, editor, *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation (SYMSAC '86)*, pages 24–30, Waterloo, Ontario, Canada, 21–23 July 1986. Association for Computing Machinery.

[2] S. Kamal Abdali, Guy W. Cherry, and Neil Soiffer. A Smalltalk system for algebraic manipulation. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 277–283, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[4] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the OOPSLA '90 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 161–168, Ottawa, Canada, 21–25 October 1990. Association for Computing Machinery. *ACM SIGPLAN Notices*, 25(10), October 1990.

[5] John Gilbert Presslie Barnes. *Programming in ADA*. Addison-Wesley, Reading, Massachusetts, 1982.

[6] Gerald Baumgartner. Type abstration using signatures. In Stallman [31], section 7.6, pages 176–177. Available as part of the GCC-2.7.0 distribution.

[7] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for C++. Technical Report CSD-TR-95-025, Department of Computer Sciences, Purdue University, August 1995.

[8] Gerald Baumgartner and Ryan D. Stansifer. A proposal to study type systems for computer algebra. RISC-Linz Report 90-87.0, Research Institute for Symbolic Computation, University of Linz, Linz, Austria, March 1990.

[9] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[10] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proceedings of the OOPSLA '89 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 457–467, New Orleans, Louisiana, 1–6 October 1989. Association for Computing Machinery. *ACM SIGPLAN Notices*, 24(10), October 1989.

[11] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, David B. MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, New York, New York, 1984. Proceedings of the International Symposium on the Semantics of Data Types, Sophia-Antipolis, France, 27–29 June 1984.

[12] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–43, August 1992.

[13] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[14] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proceedings of the OOPSLA '92 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, Vancouver, Canada, 18–22 October 1992. Association for Computing Machinery. *ACM SIGPLAN Notices*, 27(10), October 1992.

[15] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, California, 17–19 January 1990. Association for Computing Machinery.

[16] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[17] Paul Hudak (ed.), Simon Peyton Jones (ed.), Philip Wadler (ed.), Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: A non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5):Section R, May 1992.

[18] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[19] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report CSL-88-9, SRI International, 1988.

[20] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[21] Elana D. Granston and Vincent F. Russo. Signature-based polymorphism for C++. In *Proceedings of the 1991 USENIX C++ Conference*, pages 65–79, Washington, D.C., 22–25 April 1991. USENIX Association.

[22] Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System.* Springer-Verlag, New York, New York, 1992.

[23] Jacob Katzenelson, Shlomit S. Pinter, and Eugen Schenfeld. Type matching, type-graphs, and the Schanuel conjecture. *ACM Transactions on Programming Languages and Systems*, 14(4):574–588, October 1992.

[24] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An exemplar based Smalltalk. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 322–330, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[25] David B. MacQueen. Modules for Standard ML. *Polymorphism*, 2(2), October 1985.

[26] David B. MacQueen. An implementation of Standard ML modules. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 212–223, Snowbird, Utah, 25–27 July 1988. Association for Computing Machinery.

[27] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Massachusetts, 1991.

[28] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[29] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 9–16, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[30] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–45, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[31] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, Massachusetts, 14 June 1995. Available as part of the GCC-2.7.0 distribution.

[32] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the EUUG Spring '87 Conference*, Helsinki, Finnland, May 1987. European UNIX Users Group. Also available in the C++ Translator Technical Papers collection from AT&T to attendees of OOPSLA '87.

[33] Bjarne Stroustrup. Parameterized types for C++. In *Proceedings of the 1988 USENIX C++ Conference*, pages 1–18, Denver, Colorado, 17–21 October 1988. USENIX Association.

[34] Bjarne Stroustrup and Dmitry Lenkov. Run-time type identification for C++ (revised). In *Proceedings of the 1992 USENIX C++ Conference*, pages 313–339, Portland, Oregon, 10–13 August 1992. USENIX.

[35] Robert S. Sutor and Richard D. Jenks. The type inference and coercion facilities in the Scratchpad II interpreter. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 56–63, St. Paul, Minnesota, 24–26 June 1987. Association for Computing Machinery. *ACM SIGPLAN Notices*, 22(7), July 1987.

[36] Stephen M. Watt, Richard D. Jenks, Robert S. Sutor, and Barry M. Trager. The Scratchpad II type system: Domains and subdomains. In Alfonso M. Miola, editor, *Computing Tools for Scientific Problem Solving*, pages 63–82. Academic Press, London, Great Britain, 1990.

[37] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin-Heidelberg, Germany, 1985.